(12) UK Patent Application (19) GB (11) 2 351 574 (13) A

(43) Date of A Publication 03.01.2001

(21) Application No 9915142.5

(22) Date of Filing 30.06.1999

(71) Applicant(s)
International Business Machines Corporation
(Incorporated in USA - New York)
Armonk, New York 10504, United States of America

(72) Inventor(s)
Matthew Kelvin Vaughton

(74) Agent and/or Address for Service
P Waldner
IBM United Kingdom Limited, Intellectual Property Department, Hursley Park, WINCHESTER, Hampshire, SO21 2JN, United Kingdom

(51) INT CL[7]
G06F 9/45

(52) UK CL (Edition S)
G4A APL

(56) Documents Cited
US 5872977 A
Dr Dobb's /CD Release 6, Jan 88 - June 98, Husain K, "Extending imake", 1994.

(58) Field of Search
UK CL (Edition R) G4A APL
INT CL[7] G06F 9/44 9/45
Dr Dobb's Journal;
Online: COMPUTER, EPODOC, INSPEC, JAPIO, TDB, WPI

(54) Abstract Title
Multiple platform application build environment

(57) This invention relates to a build environment for multiple platforms in which a developer develops a source code application for compilation into multiple object code applications for many platforms. In an environment in which the vast majority of the source code that forms the delivered product is common across many platforms and operating system environments it is desirable to architect a build model that requires no changes in the shared source or makefile content each time an additional new operating system or platform is added to the supported list. The method of processing a build makefile within a multi platform development environment comprises the following steps. A makefile processor is instructed to execute, 103, a makefile which is common and shared across all supported platforms and environments. The common shared cross platform makefile defines only platform-independent values and then tries, 104, to include a platform-dependent makefile which only defines platform or environment specific values. The result of combining, 106, the values defined by the common shared cross-platform makefile and the platform-dependent makefile is used to build the required platform targets. The separation of platform-independent and platform-dependent build information in which the platform element is replaceable provides for development scalability, and protects the shared source code base from intrusive changes each time a new platform has to be supported.

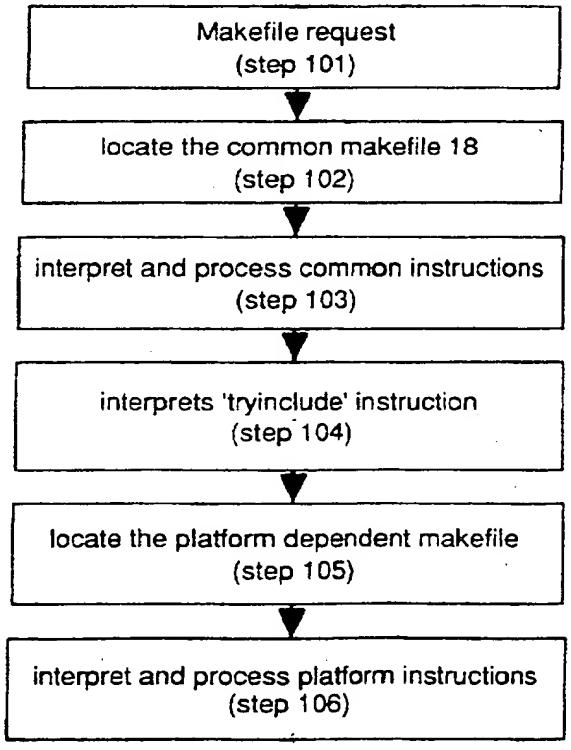

Figure 3

GB 2 351 574 A

prior art

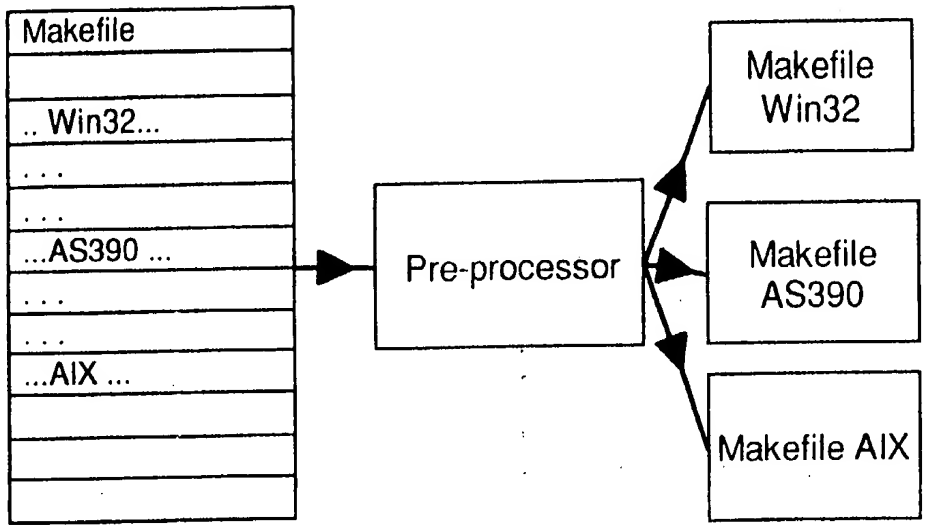

Figure 1A

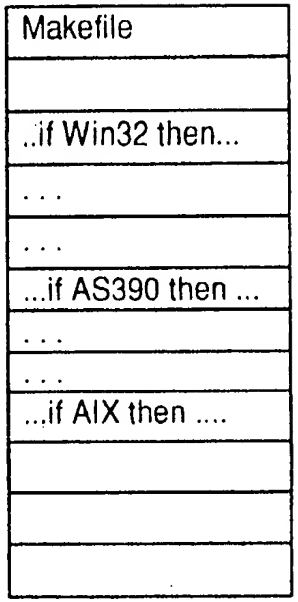

Figure 1B

prior art

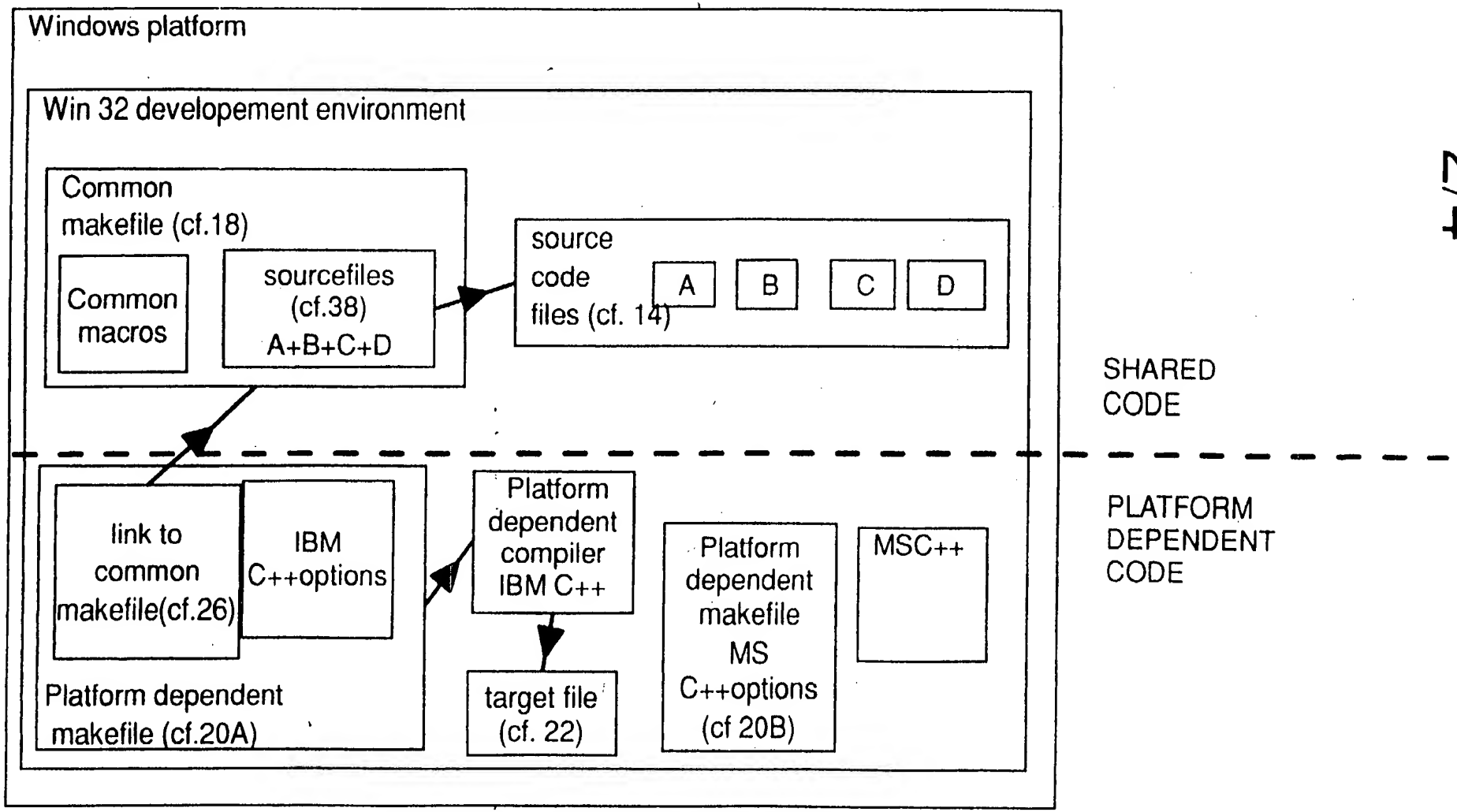

Figure 1C

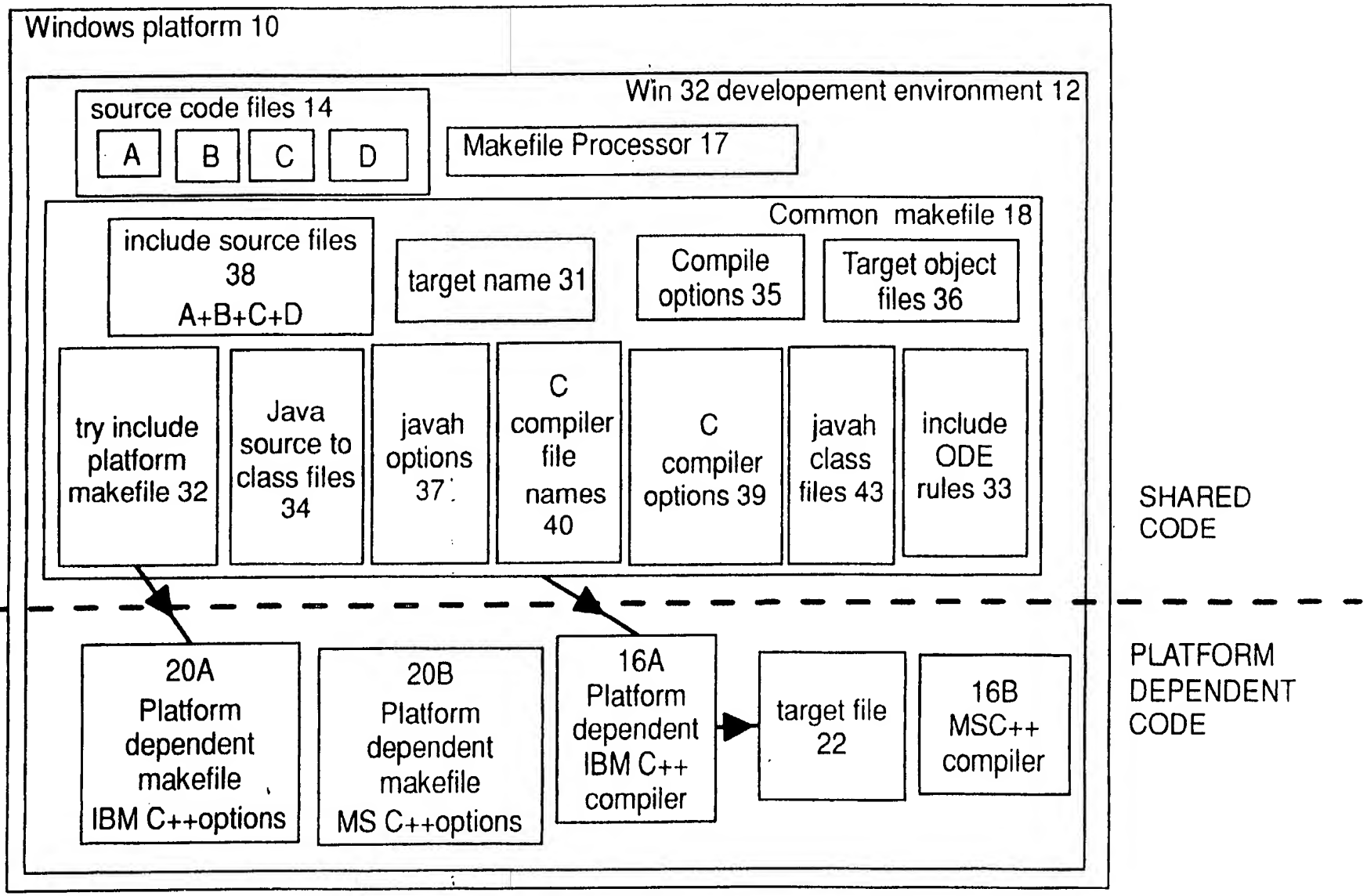
Windows platform 10
Win 32 developement environment 12
source code files 14
A
B
C
D
Makefile Processor 17
Common makefile 18
include source files 38 A+B+C+D
target name 31
Compile options 35
Target object files 36
try include platform makefile 32
Java source to class files 34
javah options 37
C compiler file names 40
C compiler options 39
javah class files 43
include ODE rules 33
SHARED CODE
PLATFORM DEPENDENT CODE
20A Platform dependent makefile IBM C++options
20B Platform dependent makefile MS C++options
16A Platform dependent IBM C++ compiler
target file 22
16B MSC++ compiler

Figure 2

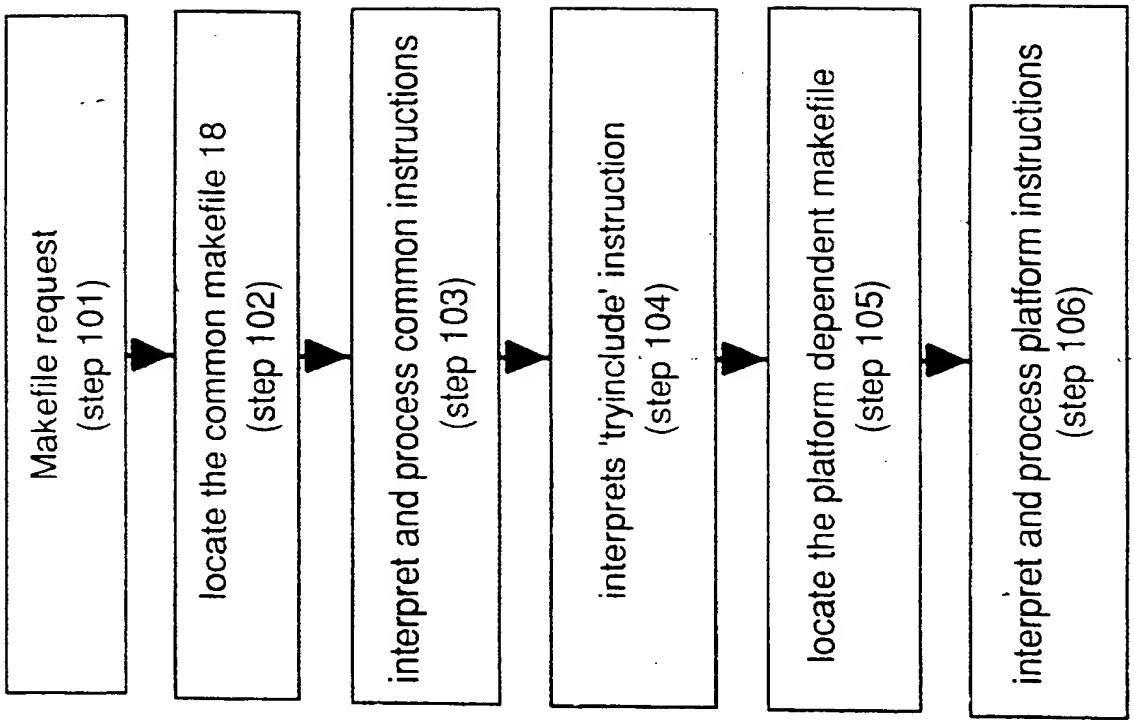
Makefile request
(step 101)
locate the common makefile 18
(step 102)
interpret and process common instructions
(step 103)
interprets 'tryinclude' instruction
(step 104)
locate the platform dependent makefile
(step 105)
interpret and process platform instructions
(step 106)

Figure 3

MULTIPLE PLATFORM BUILD ENVIRONMENT

FIELD OF INVENTION

This invention relates to a build environment for multiple platforms in which a developer develops a source code application for compilation into multiple object code applications for many platforms.

BACKGROUND OF INVENTION

In an environment in which the vast majority of the source code that forms the delivered product is common across many platforms and operating system environments it is desirable to architect a build model that requires no changes in the shared source or makefile content each time an additional new operating system or platform is added to the supported list. Traditional approaches often require platform specific make files to build both shared and platform specific code. Recent build technologies, notably the Open Software Foundation (OSF) Open Development Environment (ODE) provide make files that are common and shared across many platforms.

The IBM Open Software Foundation (OSF) Development Environment (ODE) provides a method for developers to simultaneously and independently create code for various releases of a program. This development process works in conjunction with, and does not interfere with, established releases controlled by release administrators. Developers can perform builds to test the functioning of their code against established program levels. Release administrators can use ODE to create new backing builds and, ultimately, new releases of code for completely different platforms.

Some ODE terminology is explain as follows. A project is a set of sources that have been grouped together and are treated as a unit. Projects can be a large as the OS and as small as ODE itself, each project has separate builds. A source file is a text file containing program code to be transformed, for instance, into an executable program. A target is the desired result of some transformation from source to object. Compilers and linkers are examples of some of the tools used to generate targets. Makefiles are used to define targets, and the mk command is used to update targets. The makefile is used by the mk command to specify target and dependencies and what actions to perform to create the target. In a makefile, a target file may be dependent on none or more source files.

A build is typically a process of compiling source code into object code, and then linking the object modules to create an executable

program. ODE uses its own build command and bases its mk on a building model found in most UNIX systems: the make command and a control file called makefile. makefile contains the variables and specifies the include statements needed for the build. Some basic characteristics of building with ODE are:

All shared ODE makefiles are the same cross all platforms build and mk are used for all builds mk obtain much of their information from environmental variables and command line variables. To customize the build environment, you can change these variables. Sources and built objects are maintained in separate sub directories. Headers and libraries are kept in the export subdirectory. Common makefiles contain frequently used build rules. The makefile for each component contains an INCLUDE statement for the common makefiles (.include<$ {RULES_MK}>). During a build, definitions of variables within the makefile trigger execution of rules in the common makefiles.

Makefiles contain information on how the build should be performed. One significant difference between the standard UNIX development environment and the ODE environment is ODE's use of common makefiles. Common makefiles exist in both ODE and prior art Unix - ODE's real difference is that ODE provides true cross platform compatibility. Common makefiles hold frequently-used build rules in one place so that these rules do not have to be duplicated in each individual makefile. The type of pass specified (export, comp, build, etc.), in conjunction with the variables in the makefile, determines which common rules are triggered. In ODE to include the common makefiles, you must specify the statement .include <${RULES_MK}> in the makefile for the source code to be compiled. This statement must follow all variable definitions in makefile. See ODE Build Reference for an example of a basic makefile).

While the ODE environment is an improvement over platform specific make files a further problem remains, this is the problem of managing platform specific build options and platform specific additional files without intruding platform specific requirements into the shared common make files. As the number of supported platforms and environments increases it is more and more important that platform specific requirements not intrude into the cross platform source and make files and that the cross platform source remain pure.

One known solution is that of pre-processing a single multiple platform makefile (See Figure 1A). A single 'pre-process' makefile comprising platform independent options is run through a pre-processor to generate multiple platform specific makefiles prior to use in compilation process. Such a pre-processing of a makefile is provided by a UNIX tool 'imake' further information on which can be found at

http://www.primate.wisc.edu/software/imake-stuff/imake-faq.txt. One disadvantage with this is that the 'pre-process' make file must be modified for each new platform type which creates a problem of ownership and intrusion into the shared codebase of the 'pre-process' makefile, each new development group wanting access to code developed by a previous group. Furthermore the 'pre-process' makefile increases in size with each platform with most of the code redundant for a single platform. This approach is more applicable for situations where different make processors supporting different dialects or versions are employed on each platfom/environment.

Another solution is a single makefile comprising multiple options for each platform for use in a compilation process (see Figure 1B). The compilation application knows which platform supports it and only executes options for that platform. Similar disadvantages as for the 'pre-process' makefile apply, the 'multi-option' makefile must be modified for each new platform type which creates a problem of ownership and intrusion. The 'multi-option' makefile also increases in size with each platform with most of the code redundant for each platform. This approach requires the same level of make processor functionality on all platforms - ODE can be operated in this mode.

Both the above mentioned disadvantages (scalability and intrusion) are addressed by a platform specific makefile solution written for the platform it runs on (see Figure 1C and cross referenced with Figure 2). The 'platform specific' makefile typically comprises 80% of the build instructions including make optimisations for that particular platform. The remaining 20% of the build instructions may be loaded in an included platform independent sub-Makefile containing macros.

Makeprocessor control always starts with the platform makefile containing most of the build information and the platform makefile includes a small amount of cross platform information. In many environments prior to ODE the need to employ different make processors on each platform limited the amount of information that could be contained in the shared file.

Problems with this approach are that when a new platform is introduced (i) much (typically 80%) information must be provided by the platform (ii) no 'vanilla' build can be started without the platform makefiles (iii) there is no 'optional' element to this model to allow the optional inclusion of shared information. A vanilla build is a build where no platform dependent makefile exists because it has not yet been written.

Most build information is now in the shared files, platform files are optionally included so need not exist if not needed and since control is passed to the shared makefile and not the platform make files it is possible to run 'vanilla' builds without any platform make files existing, these being added later. This is better suited to environments in which we want as little platform specific information as possible.

It is essential that a platform specific makefile is written for each new platform introduced which can slow up development for each platform. Furthermore it is necessary to choose the correct makefile to match the desired compilation. Problem with the amount of content in the platform files vs the shared files. This approach is often used when platforms have disimilar make processors as this limits the amount of shared content to the lowest common denominator level. Once common make processors are available (e.g. ODE) then the emphasis can be shifted from platform to shared, this reduces the effort to introduce each new platform and locates more data in shared files and helps reduce costs and maintenance problems.

SUMMARY OF INVENTION

In one aspect of the invention there is provided a method of processing a build within a multi-platform development environment comprising the steps of: (a) interpreting a common shared cross platform makefile using a makefile processor, said common shared cross platform makefile defines platform independent values and variables; (b) including a platform dependent makefile containing platform dependent values and variables which may add to, remove or modify previously defined platform independent values and variables; and (c) building a platform target from the resulting combined makefile values and variables.

Substituting the platform specific parts at a later stage in the makefile processing allows the support of any number of different platforms and environments without any platform requirements intruding into the pure shared cross platform source and makefiles. Any number of new platforms may be added without requiring any changes (intrusion) to the shared source code base. Development scalability is improved as any number of new platforms can be added without any requiring any changes to the shared source code & makefiles. This allows different development groups to focus on the shared and platform elements across a clean build interface. Reduced development and maintenance costs follow as a result. With much higher content (80%) now shared the cost of each new platform is reduced as the platform task is smaller.

Some differences are (i) the point where control is passed in at shared not platform (ii) optional inclusion of platform information by

shared rather than inclusion of shared by platform. Control is initially passed to the common makefile (this is in contrast to the Sun prior art model which passes control to platform specific makefiles).

The common makefile comprises a call instruction to include the platform configuration file, the call instruction to include the platform configuration file is optional. Using ODE all shared component source code makefiles use a "tryinclude" statement to optional include platform specific configuration files. The invention allows object code to be created for which there exists no platform makefile. This is possible for the embodiment because control starts with the common makefile and not the platform makefile which needs to be written first.

The step of locating the platform specific makefile comprises locating a makefile having a predefined named in a predefined location.

The common makefile may specify the name of the target file for the object code.

The common makefile may specify the names of the source code files.

The platform specific makefile preferably contains less than 30% of the total makefile instructions, the remaining instructions being included in the common makefile. In a particularly advantageous example the platform specific makefile contains less than 10% of the total makefile instructions. A further aspect of the invention uses separate shared and platform parts for each component to be built and for each of the shared parts there exists a shared part make file which tries to include an optional platform specific configuration file. If the platform specific configuration file does not exist then the build options set by the shared make files are not changed. If a platform specific configuration file exists then it can modify or add to those options set by the shared make file. Since the platform component can vary between different platforms the different platform specific configuration files can modify the shared options to their own needs with no change to the shared component.

BRIEF DESCRIPTION OF DRAWINGS

In order to promote a fuller understanding of this and other aspects of the present invention, an embodiment will now be described, by way of example only, with reference to the accompanying drawings in which:

Figure 1A, B, C are schematic representations of a first, second and third prior art solutions;

Figure 2 is schematic representation of the embodiment of the present invention; and

Figure 3 is a flow chart of the method of the present embodiment.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENT

The embodiment code base formally consists of all the of platform independent JVM source code and build makefiles files which are applicable to all supported target platforms or environments. In itself the Sovereign code base can not produce a runtime environment - it must be combined with non-Sovereign platform dependent source code and build makefiles to build a platform runtime environment.

Figure 2 is a representation of the present embodiment. Platform 10 supports Windows development environment 12 which comprises the cross platform shared source code files 14 A,B,C,D; IBM C++ compiler 16A and Microsoft C++ compiler 16B; makefile processor 17; common shared cross platform makefile 18 and platform or environment dependent makefiles 20A and 20B. Platform 10 is for instance a Microsoft Windows operating system running on an Intel Pentium processor.

Makefile processor 17 reads in the common shared cross platform makefile18 and when instructed by tryinclude 32 in the common shared cross platform makefile 18 the processor 17 reads in the platform dependent makefile 20 (A or B depending on environment) for executing the instructions contained therein in preparation for the compilation by compiler 16A or 16B of the source code files 14 into the target file 22. In this embodiment there are four cross platform shared source code files 14A,B,C,D but the number of files is not essential. There may also be platform dependent source files (not shown) within the environment.

There are two compiler components 16A and 16B for a Microsoft Windows / Intel Pentium platform. However, in a Linux platform embodiment a Linux compiler would be used, without any modification of the common makefile 18 or compulsory addition of an associated platform dependent makefile. In this embodiment several compilers, eg Microsoft Visual C/C++ & IBM C/C++ run in the same development environment 12 on the same platform 10 and there is no ability to build for other platforms. In another embodiment cross compilation (building onfor platform A on platform B) is possible and therefore further compilers and platform specific makefiles may be added to the development environment 12. The development environment can also be run on other operating systems, such as AIX or OS/390.

The common shared cross platform makefile 18 comprises an instruction 38 to define the list of common shared cross platform source

code components 14 to be built. Makefile 18 can also define other common shared cross platform definitions and values such as platform independent sources and targets and options used to build these. The common makefile further comprises an instruction 32 to include the platform generic makefile 20A or 20B. This takes the form of a 'tryinclude' instruction which attempts to include a file but does not terminate the process if no such file exists. The operand of the 'tryinclude' instruction is a predefined file name in a predefined path. For instance 'makefile_platform' in the main storage directory but another other name or directory would suffice

Referring to Figure 3 the build process starts with a request to makefile processor to process the common shared cross platform makefile so that all the common shared cross platform options and parameters are set for the build (step101). The makefile processor locates the common makefile 18 (step 102) in the development environment and interprets and performs the instructions contained therein (step 103). When the makefile processor 17 comes to a 'tryinclude makefile_platform' instruction (step 104) it locates the platform dependent makefile 20A (step 105) and begins to interpret and process instructions contained therein (step 106).

Non-sovereign, platform dependent code is integrated into the build through a well defined extension model which allows platforms to influence the build without having to change or invade the Sovereign code space.

In order to build some platform objects/targets ... it may be necessary for platforms to add or include some platform objects with the Sovereign objects, in addition it is usual for the platform to want to influence the compile and link options used during the build. Since the names of additional objects, compile and link options is platform dependent it is necessary to allow the platform to set these in collaboration with the shared Sovereign parts of the build with invading the Sovereign code space. The platform is allowed to set its own options by means of a 'tryinclude' statement in the bottom of each Sovereign component makefile. If no optional platform configuration file exists in the platform directory (pfm) then no settings are changed.

This is an example of a component makefile of the embodiment:

```
SHARED_LIBRARIES = ${SHLIB_PREF}jvm${MY_TYPE}${SHLIB_SUFF} [31]
JAVASRC = InvokerGen.java [34]
JFLAGS = -J-ms16m -J-mx64m -nowarn -bootclasspath "" [35]
OFILES = ${MY_SOVEREIGN_OBJS} [36]
OFILES += invokers${OBJ_SUFF}
JAVAHFLAGS += -old [37]
```

```
JAVAHSRC = \ [43]
java.io.InputStream \
java.lang.Boolean \
java.lang.Byte \
sun.misc.VM
CFLAGS += -DHPROF -DCHECK_JNI -DBREAKPTS -DDELAY_JIT_LOADING [39]
INCFLAGS += $(HPIHEXPS) $(JVMHEXPS) [40]
.if ( ($(MY_TYPE) == "_d") || ($(MY_TYPE) == "_g") )
CFLAGS += -DLOGGING -DTRACING -DJCONV
.else
CFLAGS += -DDEBUG
.endif
invokers.c: InvokerGen.class [41]
$(JAVA) -classpath $(CLASSPATH) InvokerGen <
$(PATH2MY_SOVEREIGN)$(DIRSEP)invokers.txt > invok- ers.c
.tryinclude <$(MAKEFILE_PLATFORM)> [32]
.include <$(RULES_MK)> [33]

[31] target to be built
[32] call to include optional platform configuration file
[33] required call to ODE rules file
[34] Java source to be compiled into class files
[35] Options to be passed to javac when java source is compiled
[36] Object files used to construct target
[37] Options to be passed to Java when header files are generated
[39] Options to be passed to the C compiler
[40] Include file directories to be passed to the C compiler
[41] Additional rules
[43] class files for which header files should be generated with Java
```

A typical optional platform configuration file would be included by the statement numbered [2] above, it would reside in platform directory and be called makefile.platform. Remember that makefile.platform files are always included by a Sovereign component make file - they are never executed stand alone. An example of a typical platform configuration file follows with referenced comments:

```
OFILES += $(MY_PLATFORM_OBJS)[51]
OFILES += invokeNative_x86.obj
CFLAGS += -WX [52]
VPATH += $(PATH2MY_PLATFORM) [53]
SHLDFLAGS += -export:jio_vfprintf -export:jio_vsnprintf [54]
invokeNative_x86.obj: invokeNative_x86.asm [55]
ml -nologo -Fl -Zi -coff -c
$(PATH2MY_PLATFORM)$(DIRSEP)invokeNative_x86.asm
```

[51] Append additional platform objects being added to the list of objects to build (note: use of += to append rather than assign)
[52] Append additional platform C compiler flags
[53] Since additional objects have been added to OFILES and these objects reside in the platform directory we need to include the platform directory in the search path
[54] Append additional platform linker shared library flags
[55] Define a new rule for building a platform specific object

In summary there is described a build environment for multiple platforms in which a developer develops a source code application for compilation into multiple object code applications for many platforms.. In an environment in which the vast majority of the source code that forms the delivered product is common across many platforms and operating system environments it is desirable to architect a build model that requires no changes in the shared source or makefile content each time an additional new operating system or platform is added to the supported list. The method of processing a build makefile within a multi platform development environment comprises of the following steps. A makefile processor is instructed to execute a makefile which is common and shared across all supported platforms and environments. The common shared cross platform makefile defines only platform independent values and then tries to include a platform dependent makefile which only defines platform or environment specific values. The result of combining the values defined by the common shared cross platform makefile and the platform dependent makefile is used to build the required platform targets. The separation of platform independent and platform dependent build information in which the platform element is replaceable provides for development scalability, and protects the shared source code base from intrusive changes each time a new platform has to be supported.

Now that the invention has been described by way of a preferred embodiment, various modifications and improvements will occur to those person skilled in the art. Therefore it should be understood that the preferred embodiment has been provided as an example and not as a limitation.

CLAIMS

1. A method of processing a build within a multi-platform development environment comprising the steps of:
(a) interpretting a common shared cross platform makefile using a makefile processor, said common shared cross platform makefile defines platform independent values and variables;
(b) including a platform dependent makefile containing platform dependent values and variables which may add to, remove or modify previously defined platform independent values and variables; and
(c) building a platform target from the resulting combined makefile values and variables.

2. A method as in claim 1 wherein the platform dependent makefile is located as instructed by the common makefile.

3. A method as in claim 2 wherein the platform dependent makefile is located by performing an instruction in the common makefile to include the specific platform makefile.

4. A method as in claim 3 further comprising performing a 'tryinclude' instruction which attempts to include a file if such a file exists but skips to the next instruction if no such file exists or such file can not be included.

5. A method as in any of claims 1 to 4 wherein the step of locating the platform specific makefile comprises locating a makefile having a predefined name in a predefined location.

6. A method as in any of the preceding claims further comprising locating the name of the target file for the object code in the common makefile.

7. A method as in any of the preceding claims further comprising locating the names of the source code files in the common makefile.

8. A method as in any of the preceding claims further comprising locating the majority of the makefile instructions in the common makefile.

9. A method as in claim 8 further comprising locating less than 30% of the total makefile instructions in the specific makefile.

10. A method as in claim 10 comprising locating no makefile instructions in the specific makefile

11. A system for processing a build makefile within a multiplatform development environment comprising : means for locating a common shared cross platform makefile associated with the development environment;

means for performing platform independent instructions contained in a common makefile;

means for locating a platform specific makefile associated with the specific platform compilation; and

means for performing platform dependent instructions in that specific makefile.

The Patent Office

INVESTOR IN PEOPLE

Application No: GB 9915142.5
Claims searched: 1-11

Examiner: Geoff Western
Date of search: 2 February 2000

Patents Act 1977
Search Report under Section 17

Databases searched:

UK Patent Office collections, including GB, EP, WO & US patent specifications, in:

UK Cl (Ed.R): G4A (APL)

Int Cl (Ed.7): G06F 9/44 9/45

Other: Dr Dobb's Journal ;
Online : COMPUTER, EPODOC, INSPEC, JAPIO, TDB, WPI

Documents considered to be relevant:

Category	Identity of document and relevant passage	Relevant to claims
X	US 5872977 A (IBM, THOMPSON J A) See whole document	11
A	Dr Dobb's /CD Release 6, Jan '88 - June '98, Husain K, "Extending imake", 1994. See whole document	-

X	Document indicating lack of novelty or inventive step	A	Document indicating technological background and/or state of the art.
Y	Document indicating lack of inventive step if combined with one or more other documents of same category.	P	Document published on or after the declared priority date but before the filing date of this invention.
&	Member of the same patent family	E	Patent document published on or after, but with priority date earlier than, the filing date of this application.